GramFuzz: Fuzzing Testing of Web Browsers Based on Grammar Analysis and Structural Mutation

Tao Guo, Puhan Zhang, Xin Wang
China Information Technology Security valuation Center
Beijing, China
{guot, hangph}@itsec.gov.cn,wx_itsec@163.com

Qiang Wei
State Key Laboratory of Mathematical Engineerning and Advanced Computing
Zhenzhou, China
chinapkuwq@gmail.com

***Abstract*—Fuzz testing is an automated black-box testing technique providing random data as input to a software system in the hope to find vulnerability. In order to be effective, the fuzzed input must be common enough to pass elementary consistency checks. Web Browser accepts JavaScript, CSS files as well as the html as input, which must be considered in fuzzing testing, while traditional fuzzing technology generates test cases using simple mutation strategies, ignoring the grammar and code structure .In this article, vulnerability patterns are summarized and a new fuzzing testing method are proposed based on grammar analysis of input data and mutation of code structure. Combining methods of generation and mutation, test cases will be more effective in the fuzzing testing of web browsers. Applied on the Mozilla and IE web browsers, it discovered a total of 36 new severe vulnerabilities(and thus became one of the top security bug bounty collectors within this period).**

Keywords-fuzzing testing; web brwoser; grammar analysis; structural mutationt

I. INTRODUCTION[1]

Software security [1] issues are expensive and risky. Security testing employs a mix of techniques to find vulnerabilities in software. One of these techniques is fuzzing[2-6] testing—a process that automatically generates random data input. Crashes or unexpected behavior point to potential software vulnerabilities[7, 8].

In traditional fuzzing testing technologies, there are two methods in producing the test cases, respectively based on generation[9-12] and mutation[13,14]. The generation method usually gets all testing interfaces and relevant parameters, then generate the test cases based on certain rules, such as jsfunfuzz[13]. In this method, knowledge about the system under test is essential, which is difficult for testers. On the other hand, most of the test cases don't satisfy the grammar standard, which will be abandoned by the browser interpreter, leading to low efficiency.

The mutation method starts from an original test case, getting various code snippets, combining them randomly and getting new test cases, such as LangFuzz[15].

This paper is supported by Natural Science foundation of China {NO.61272493, NO.61100047} and National High Technology Research and Development Program (863 Program) of China {NO.2012AA012903}.

This method solves the problem that test cases which do not satisfy the grammar standard will be abandoned by browser interpreter and gets more effective testing cases. While LangFuzz only focus on JavaScript, ignoring the HTML tag, CSS files and Dom objects, which are also processed by the browsers. Although this method reduces the blindness in fuzzing testing, there are a lot of limitations. For complex vulnerabilities, this method isn't very effective[16].

To solve this problem, a new method of fuzzing testing is proposed in this article. Getting web files on the Internet by gatherers, analyze the source codes in HTML, CSS and JavaScript files, build the grammar trees and code snippets respectively, and mutate the test cases with structural strategies. With this method, fuzzing testing will be more effective and unexposed vulnerabilities will be found.

II. ANALYSIS OF VULNERABLITY PATTERNS

In recent years, zero-day vulnerabilities of web browsers, such as IE and Firefox, have been increased rapidly and researchers pay more attention to the security testing for web browsers.

While traditional testing technology only focus on the testing of JavaScript. From the latest vulnerabilities, we can see that the codes in POC are all very complex, including DOM objects, CSS style, and JavaScript. Use-after-free, buffer overflow of stack and heap are the main cause of vulnerabilities.

To trigger these vulnerabilities, HTML tags, CSS style, JavaScript code must be included in the POC file through a complex combination. It is a problem that how to build a test case which can trigger underlying problems usually ignored by traditional testing.

To analyze these vulnerabilities, let's start with a real case.

```
CVE-2010-3971
// html file
<div style="position: absolute; top: -999px;left: -999px;">
<link href="css.css" rel="stylesheet" type="text/css" />
// css file
*{
    color:red;
}
@import url("css.css");
@import url("css.css");
@import url("css.css");
@import url("css.css");
```

Figure 1. Trigger code of CVE-2010-3971.

Fig. 1 shows the trigger code of CVE-2010-3971 vulnerability. The first two lines declare a CSS style and point to the file. In the file, a CSS style is defined in the following three lines. The next 4 lines refer the CSS style and trigger a vulnerability of memory corruption.

```
MS10-018
<style type="text/css">
.y{behavior: url(#default#userData);}
</style>
<marquee vyxixo="[object HTMLCollection]"
        id="x" class="y">
</marquee>
for (i= 1; i<10; i++ ){
            x.setAttribute("vyxIxO",document.location);
}
x.setAttribute("vyxIxO",document.getElementsByName("style"));
document.location="about:\u0c0c\u0c0c\u0c0c\u0c0c\u0c0cblank";
```

Figure 2. Trigger code of MS10-018.

Fig. 2 shows the trigger code of MS10-018 vulnerability. The first three lines declare a CSS style and apply it in a HTML file. The next three lines declare a HTML Collection object. In the last five lines, a series of complex operations are applied on the objects in current HTML pages. These operations trigger the vulnerability.

In traditional fuzzing testing technologies, it is hard to generate test cases which not only satisfy the grammar standard, but also covers a lot on the HTML objects, CSS styles and JavaScript, due to the lack of concern on JavaScript grammar, CSS style and HTML Dom objects. In this paper, we propose a new method of generating test cases, analyze the html files by the gatherers, build the grammar tree and test snippets library. Slice the test codes, generating test cases with a special method by combination of generation and mutation to solve this problem.

This method has three advantages below:

- The testers do not need a deep knowledge on DOM objects, CSS style and JavaScript grammar; just get them from the code snippet library.
- The code from real HTML files will lead to a high coverage of test interfaces, which will make the fuzzing testing more effective.
- The method of generating test cases aims at the structure of grammar tree, gives the test cases more capability on deep underlying vulnerabilities.

III. IMPLEMENTATION OF GRAMFUZZ

A. Structure

To get more train sets and initial test cases, GramFuzz uses gatherers to get HTML files from the Internet and build the train sets and initial test cases. Then we analyze the grammar of the train sets; build the grammar library of JavaScript, CSS and HTML files. After that, we use files and grammar nodes in the library as the basic element to mutate and generate the new test cases. The whole program structure is shown in Fig. 3.

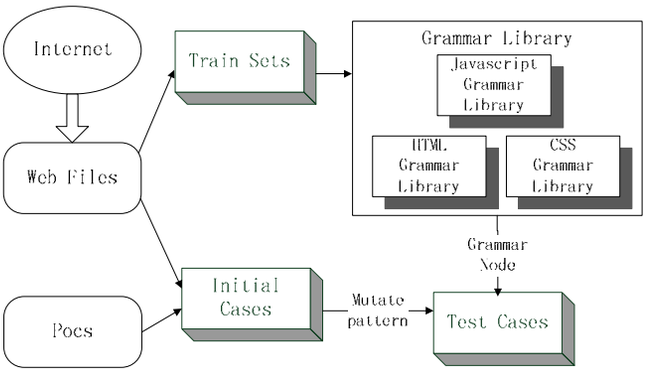

Figure 3. Trigger code of MS10-018.

B. Build the grammar library

GramFuzz gathers Web files from the Internet, chooses the larger-sized files as train sets to get a richer library. The chosen files are separated to three parts to deal with as the different grammars: HTML, CSS and JavaScript. These three languages constitute the Web pages.

To address these three kinds of train sets, we analyze with the corresponding grammar, getting the grammar tree and preparing for the following steps.

Fig. 4 shows the HTML and CSS Grammar Specification. HTML is a kind of mark language, not a programming language. Tags are used to descript the contents in HTML, which are put in coupled tags. By analyzing the tags, we can get the structure of HTML text in a grammar tree, which is constructed by a root node and numbers of tag child node and shown as Fig. 4.

```
HTML Grammar Specification
<HTML> ::= <Tags>
<Tags> ::= <Tags> <Tag>
<Tag> ::= </TagStart> <Content> </TagEnd>
CSS Grammar Specification
<CSS> ::= <Selectors>
<Selectors> ::= <Selectors> <Selector>
<Selector> ::= <Declarations>
<Declarations> ::= <Declarations> <Declaration>
<Declaration> ::= <Property> ; <Value>
```

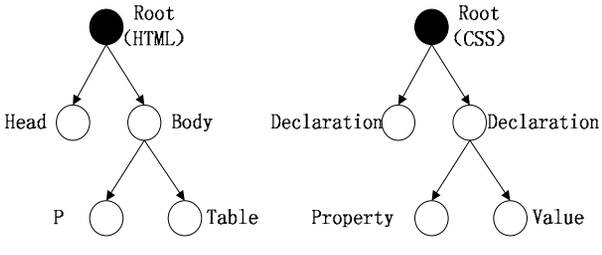

Figure 4. HTML and CSS Grammar Tree

```
Javascript Grammar Specification：
<Program> ::= <Source Elements>
<Source Elements> ::= <Source Elements><Source Element>
<Source Element> ::= <Statement>
            | <Function Declaration>
<Function Declaration> ::= 'function' Identifier '(' <Formal Parameter List> ')'
                '{' <Function Body> '}'
              | 'function' Identifier '(' ')' '{' <Function Body> '}'
<Statement>   ::= <Block>
          | <Assignment Statement>
          | <Variable Statement>
          ......
```

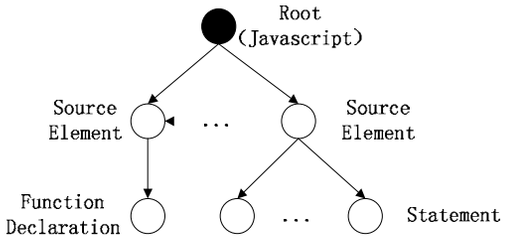

Figure 5. Javascript Grammar Tree

CSS styles are constructed by two parts: selectors and one or more definitions. Selectors are the HTML elements which need to change their style. Definitions contain a property and a style attribute. Every property has an attribute, separated by a colon. A CSS grammar tree is constructed by a root node and numbers of selector child node, which is shown in Fig. 4.

Grammar of JavaScript is a little more complicated. Fig. 5 shows a part of the grammar of JavaScript. JavaScript language is constructed by a lot of source elements, each element could be a function declaration or statement. A JavaScript grammar tree contains a program root node and numbers of source elements child nodes. Keep dividing from the child nodes in the grammar tree; we could get some non-terminator nodes and terminator leaf nodes, which is shown in Fig. 5.

C. *Extract the grammar node*

Grammar nodes are the nodes in the generated grammar tree. After the grammar tree is built, we can extract the grammar nodes and put them in the library for the following steps in generating test cases. This is an important part in the whole program.

Grammar nodes consist of a tuple-element: Type & Value. Type is the node type of every child node; Value means the content of every child node, the code snippet. We use a depth-first search algorithm to search the grammar tree from the root node, put every child node in the library with their type value. The algorithm is shown in Fig. 6.

```
Input：RootNode
Output：GramNode_DataBase

Node CurrentNode=RootNode;
Void SearchLeafNode(Node n) {
   Node temp = new Node();
   Temp.code = n.code;
   Temp.type = n.type;
   GramNode_DataBase.pushNode(temp);
}
Void SearchChildNode(Node n) {
   SearchLeafNode(n);
   Node temp = n.nextLeafNode();
   While(temp!=NULL) {
        SearchChildNode(temp);
        temp = n.nextLeafNode();
   }
}
Void Main(Node RootNode) {
   SearchChildNode(RootNode);
}
```

Figure 6. Algorithm of Extracting the Grammar Node

D. *Generate Test Cases*

Initial test cases are the template of generating new test cases, it is crucial to build a build a proper initial test case. When choosing the initial cases, the following tips are very important:

- Don't choose files which has a very large size, otherwise the generating program will be slow and many cases will be insignificant.
- The initial cases should contain plenty JavaScript files and CSS styles, with a complicated structure and function calls. This will make the cases generated more complicated and more effective in testing.

Thus, we gather files from the Internet, and choose the initial test cases. Then we analyze the initial cases and get their grammar structure. After that, we can mutate the initial cases with a certain pattern, approaching a structural grammar testing.

From the root node of initial case, we search the grammar tree in a depth-first method, and mutate each grammar node with the following steps in a probability of P:

- Substitute current node with random type of node in the library
- Duplicate current node to another random place
- Delete current node
- Mutate the content of current node, including abusing the code, substitute an abnormal integer or string
- Remain current node unchanged

There is no explicit according to choose the probability of P, usually we fix it in the real testing. If P is set larger, the mutation will be more intensive and vice-versa.

IV. EXPERIMENT

The grammar analysis of web pages that fetched from the Internet is completed by Gold Parser, an open source analysis tool. Gold Parser has finished the back-end of compilation, and works very well with the front-end lexical and grammar features of JavaScript, CSS and HTML given by us. When the analysis finishes, we can get the grammar tree and generate the test cases.

To verify the effect and compare with traditional tools such as jsfunfuzz and LangFuzz, real testing for IE, Firefox and Mozilla in various versions is taken in this article. The result is shown in table. 1.

TABLE I. TESTING RESULT

	Testing Tools		
	Jsfunfuzz	***LangFuzz***	***GramFuzz***
Test Cases (Thousand)	100	80	150
Test Time (hour)	100	100	100
Bugs	25	20	36

Comparing the bugs found by different tools, we can evaluate the effect of the tools. The bugs found by different tools are not all the same cause the different testing method, but GramFuzz find the most bugs and many of which are not found by previous tools. The comparison is shown in fig. 7.

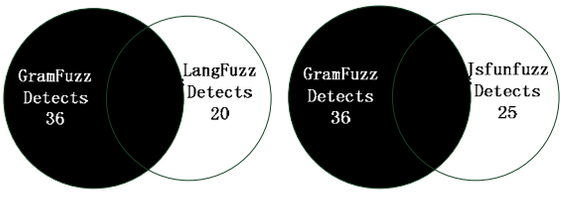

Figure 7. Algorithm of Extracting the Grammar Node

Consider the 21 defects that were exclusively detected by GramFuzz compared to LangFuzz and 14 defects compared to jsfunfuzz, the result prove that GramFuzz is capable to find bugs while previous tools can't.

V. CONCLUSION

In this article, a new testing method of web browsers is proposed. Considering the HTML, CSS styles and JavaScript in web pages, using grammar analysis to get the grammar tree, mutation the test cases in grammar structure, substitute the code snippet with the codes in library to generate more effective test cases. Applied to real testing, the result proves the effect and capability of GramFuzz, and find out bugs that was previously undiscovered.

REFERENCES

[1] Holler C, Herzig K, Zeller A. Fuzzing with code fragments[C]//Proc. USENIX Security. 2012: 445-458.

[2] The phpparser project. http://code.google.com/p/phpparser/.

[3] AITEL, D. The advantages of block-based protocol analysis for security testing. Tech. rep., 2002.

[4] Godefroid P, Kiezun A, Levin M Y. Grammar-based whitebox fuzzing[C]//ACM SIGPLAN Notices. ACM, 2008, 43(6): 206-215.

[5] Lindig C. Random testing of C calling conventions[C]//Proceedings of the sixth international symposium on Automated analysis-driven debugging. ACM, 2005: 3-12.

[6] Miller B P, Fredriksen L, So B. An empirical study of the reliability of UNIX utilities[J]. Communications of the ACM, 1990, 33(12): 32-44.

[7] Miller C, Peterson Z N J. Analysis of mutation and generation-based fuzzing[J]. White Paper, Independent Security Evaluators, Baltimore, Maryland (securityevaluators. com/files/papers/analysisfuzzing. pdf), 2007.

[8] Molnar D, Li X C, Wagner D A. Dynamic test generation to find integer bugs in x86 binary linux programs[C]//Proceedings of the 18th conference on USENIX security symposium. USENIX Association, 2009: 67-82.

[9] Neuhaus S, Zimmermann T, Holler C, et al. Predicting vulnerable software components[C]//Proceedings of the 14th ACM conference on Computer and communications security. ACM, 2007: 529-540.

[10] Oehlert P. Violating assumptions with fuzzing[J]. Security & Privacy, IEEE, 2005, 3(2): 58-62.

[11] Parr T J, Quong R W. ANTLR: A predicated‐LL (k) parser generator[J]. Software: Practice and Experience, 1995, 25(7): 789-810.

[12] Purdom P. A sentence generator for testing parsers[J]. BIT Numerical Mathematics, 1972, 12(3): 366-375.

[13] RUDERMAN, J. Introducing jsfunfuzz.

http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/, 2007.

[14] Shu G, Hsu Y, Lee D. Detecting communication protocol security flaws by formal fuzz testing and machine learning[M]//Formal Techniques for Networked and Distributed Systems–FORTE 2008. Springer Berlin Heidelberg, 2008: 299-304.

[15] Sutton M, Greene A. The art of file format fuzzing[C]//Blackhat USA Conference. 2005.

[16] SUTTON, M., GREENE, A., AND AMINI, P. Fuzzing: Brute Force Vulnerability Discovery. Addison-Wesley Professional, 2007.